第二次上机报告(lab_03)

3.1 了解如何编程实现进程创建

基础概念

- **进程**：包含代码、数据和系统分配的资源
- **fork()**：系统调用创建与当前进程几乎完全相同的副本（子进程）
- 执行特点：
  - 被调用一次，返回两次（父/子进程各一次）
  - 返回值：
    - 父进程：返回子进程PID
    - 子进程：返回0
    - 错误：返回负值

进程关系

- 父子进程形成链表关系：
  - 父进程的fpid指向子进程ID
  - 子进程的fpid为0
- 每个进程有唯一PID（getpid()）和父进程PID（getppid()）

变量特性

- fork后变量独立存在于不同地址空间
- 复制的是进程当前状态，不是从头开始执行

循环中的fork

- 每次fork会产生指数级增长的进程
- 计算公式：
  - 子进程数：1+2+4+...+2^(N-1)
  - printf执行次数：2*(1+2+4+...+2^(N-1))

孤儿进程

- 父进程终止后，子进程会被init进程（PID=1）接管

printf缓冲问题

- 带`\n`的printf会立即刷新缓冲区
- 不带`\n`的printf会被子进程继承缓冲区内容

逻辑运算符影响

- `&&`和`||`会影响fork的执行路径：
  - A&&B：A为0时不执行B
  - A||B：A非0时不执行B
- 复杂的逻辑表达式会产生更多分支进程

进程数计算

- 通过分析逻辑运算的分支路径计算总进程数
- 示例：特定模式可产生20个进程（含main）

3.2 输入代码，观察输出结果(fork1)

使用命令`mkdir os_lab_3`创建文件夹，并使用`touch fork1.c`创建 C 语言脚本，通过vim命令输入如下命令

```c
#include <stdio.h>
#include <sys/types.h>
// #include <unistd.h>
#include <stdlib.h>

int value = 5;  // where? 全局变量

int main() {
    int i;  // where? 主函数中

    pid_t pid;

    for (i = 0; i < 2; i++) {  // How many new processes and printfs?

        pid = fork();

        if (pid == 0) {
            value += 15;
            printf("Child: value = %d\n", value);
        } else if (pid > 0) {
            wait(NULL);
            printf("PARNET: value = %d\n", value);
            exit(0);  // Notice: What will happen with or without this line?
        }
    }
}
```

使用`gcc -o fork1 fork1.c`编译源码，通过`./fork1`得到如下结果:

```
kpmark@LAPTOP-AITJPLGC:/mnt/d/os-homework/os_lab_3$ ./fork1
Child: value = 20
Child: value = 35
PARNET: value = 20
PARNET: value = 5
```

执行流程分析：

1. 最初状态
    - 父进程 P 开始执行，value = 5
    - i = 0
2. 第一次循环 (i=0)
    - P 调用 fork() 创建子进程 C1
    - C1 中 value 增加 15，变成 20，打印："Child: value = 20"
    - P 等待 C1 结束
3. C1 进入第二次循环 (i=1)
    - C1 调用 fork() 创建子进程 C2
    - C2 中 value 从 20 再增加 15，变成 35，打印："Child: value = 35"
    - C1 等待 C2 结束，然后打印："PARNET: value = 20"（注意这里是 C1 作为父进程打印的）
    - C1 执行 exit(0) 结束
4. 最初的父进程 P 继续
    - C1 已结束，P 继续执行并打印："PARNET: value = 5"
    - P 执行 exit(0) 结束

如果删除`exit(0)`，则结果变为:

```
kpmark@LAPTOP-AITJPLGC:/mnt/d/os-homework/os_lab_3$ ./fork1
Child: value = 20
Child: value = 35
PARNET: value = 20
PARNET: value = 5
Child: value = 20
PARNET: value = 5
```

最初状态:

- 原始父进程P开始执行，value = 5
- i = 0（第一次循环）

第一次循环 (i=0):

- P调用fork()创建子进程C1
- C1中value增加15，变成20，打印："Child: value = 20"
- P等待C1结束

C1进入第二次循环 (i=1):

- C1调用fork()创建子进程C2
- C2中value从20再增加15，变成35，打印："Child: value = 35"
- C1等待C2结束，然后打印："PARNET: value = 20"（C1作为父进程）
- C1循环结束（没有exit所以不会终止）

最初的父进程P继续:

- C1已结束其循环，P继续执行并打印："PARNET: value = 5"
- P进入第二次循环(i=1)

第二次循环 (i=1, P进程):

- P调用fork()创建另一个子进程C3
- C3中value增加15，变成20，打印："Child: value = 20"

- P等待C3结束，然后打印："PARNET: value = 5"
- P循环结束，程序终止

3.3 掌握如何通过管道实现进程间通信

3.3.1 阅读示例的代码，编译执行，并加以理解

在根目录下创建`ppipe.c`文件，并写入以下内容:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char* argv[]) {
    int pid;
    int pipe1[2];
    int pipe2[2];
    int x;
    if (pipe(pipe1) < 0) {
        perror("failed to create pipe1");
        exit(EXIT_FAILURE);
    }
    if (pipe(pipe2) < 0) {
        perror("failed to create pipe2");
        exit(EXIT_FAILURE);
    }
    pid = fork();
    if (pid < 0) {
        perror("failed to create new process");
        exit(EXIT_FAILURE);
    } else if (pid == 0) {
        // 子进程=>父进程：子进程通过pipe2[1]进行写
        // 子进程<=父进程：子进程通过pipe1[0]读
        // 因此，在子进程中将pipe1[1]和pipe2[0]关闭
        close(pipe1[1]);
        close(pipe2[0]);
        do {
            read(pipe1[0], &x, sizeof(int));
            printf("child %d read: %d\n", getpid(), x++);
            write(pipe2[1], &x, sizeof(int));
        } while (x <= 9);
        close(pipe1[0]);
        close(pipe2[1]);
    } else {
        // 父进程<=子进程:父进程从pipe2[0]读取子进程传过来的数
        // 父进程=>子进程:父进程将更新的值通过pipe1[1]写入，传给子进程
        // 因此，父进程会先关闭pipe1[0]和pipe2[1]端口
        close(pipe1[0]);
        close(pipe2[1]);
        x = 1;
        do {
            write(pipe1[1], &x, sizeof(int));
            read(pipe2[0], &x, sizeof(int));
            printf("parent %d read: %d\n", getpid(), x++);
        } while (x <= 9);
        close(pipe1[1]);
        close(pipe2[0]);
    }
```

```
    return EXIT_SUCCESS;
}
```

然后编写`Makefile`，写入以下内容:

```
srcs=ppipe.c
objs=ppipe.o
opts=-g -c
all:ppipe
ppipe: $(objs)
    gcc $(objs) -o ppipe
ppipe.o: $(srcs)
    gcc $(opts) $(srcs)
clean:
    rm ppipe *.o
```

使用`make`命令即可默认执行`all`选项进行编译，使用`./ppipe`命令执行结果如下:

```
kpmark@LAPTOP-AITJPLGC:/mnt/d/os-homework/os_lab_3$ ./ppipe
child 16469 read: 1
parent 16468 read: 2
child 16469 read: 3
parent 16468 read: 4
child 16469 read: 5
parent 16468 read: 6
child 16469 read: 7
parent 16468 read: 8
child 16469 read: 9
parent 16468 read: 10
```

执行流程:

1. **初始状态**：x = 1（在父进程中）
2. 父进程
    - 写入x=1到pipe1
    - 从pipe2读取子进程处理后的数值
    - 输出读取到的值，并将x加1
    - 循环直到x > 9
3. 子进程
    - 从pipe1读取父进程传来的数值
    - 输出读取到的值，并将x加1
    - 将修改后的x写入pipe2传给父进程
    - 循环直到x > 9

3.3.2 完成独立实验

使用如下代码完成题目要求:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
```

```c
#include <sys/wait.h>
#include <unistd.h>

int calculate_fx(int x) {  // f(x)
    if (x == 1)
        return 1;
    return calculate_fx(x - 1) * x;
}

int calculate_fy(int y) {  // f(y)
    if (y == 1 || y == 2)
        return 1;
    return calculate_fy(y - 1) + calculate_fy(y - 2);
}

int main(int argc, char* argv[]) {
    int x = atoi(argv[1]);
    int y = atoi(argv[2]);

    int pipe_fx[2];  // f(x)进程向主进程传值
    int pipe_fy[2];  // f(y)进程向主进程传值

    // 创建管道
    if (pipe(pipe_fx) < 0 || pipe(pipe_fy) < 0) {
        perror("pipe error");
        exit(1);
    }

    pid_t pid_fx = fork();

    if (pid_fx < 0) {
        perror("fork error");
        exit(1);
    } else if (pid_fx == 0) {
        // 子进程计算f(x)
        close(pipe_fx[0]);  // 0读1写
        close(pipe_fy[0]);
        close(pipe_fy[1]);

        int result_x = calculate_fx(x);
        printf("进程ID %d 计算 f(%d) = %d\n", getpid(), x, result_x);

        write(pipe_fx[1], &result_x, sizeof(result_x));
        close(pipe_fx[1]);

        exit(0);
    }

    pid_t pid_fy = fork();

    if (pid_fy < 0) {
        perror("fork error");
        exit(1);
    } else if (pid_fy == 0) {
        // 子进程计算f(y)
        close(pipe_fy[0]);
        close(pipe_fx[0]);
        close(pipe_fx[1]);
```

```
        int result_y = calculate_fy(y);
        printf("进程ID %d 计算 f(%d) = %d\n", getpid(), y, result_y);

        // 将结果写入管道
        write(pipe_fy[1], &result_y, sizeof(result_y));
        close(pipe_fy[1]);

        exit(0);
    }

    close(pipe_fx[1]);
    close(pipe_fy[1]);

    int result_x, result_y;

    read(pipe_fx[0], &result_x, sizeof(result_x));

    read(pipe_fy[0], &result_y, sizeof(result_y));

    close(pipe_fx[0]);
    close(pipe_fy[0]);

    int result_xy = result_x + result_y;

    // 等待子进程结束，前面没资源的自己会阻塞
    waitpid(pid_fx, NULL, 0);
    waitpid(pid_fy, NULL, 0);

    printf("进程ID %d 计算 f(%d,%d) = f(%d) + f(%d) = %d + %d = %d\n", getpid(),
x, y, x, y, result_x, result_y, result_xy);

    return 0;
}
```

运行结果如下:

```
kpmark@LAPTOP-AITJPLGC:/mnt/d/os-homework/os_lab_3$ ./fork2 6 7
进程ID 18218 计算 f(6) = 720
进程ID 18219 计算 f(7) = 13
进程ID 18217 计算 f(6,7) = f(6) + f(7) = 720 + 13 = 733
```

Git 记录

```
PS D:\os-homework> git log
commit 9b893924ceab19b94c5154c95d3244f952b30adf (HEAD -> main, origin/main)
Author: kpmark <2585050765@qq.com>
Date:   Tue May 20 19:36:16 2025 +0800

    新增上机报告

commit 606550fdc8d288728cd730da1bc9e6c3dcf60cad
Author: kpmark <2585050765@qq.com>
Date:   Tue May 20 18:45:15 2025 +0800

    新增fork2代码

commit 275833e35b1bf031c252b46733b7ddff0e7441de
Author: kpmark <2585050765@qq.com>
Date:   Tue May 20 18:25:16 2025 +0800

    新增实验代码
```

os-homework/os_lab_3 at main · markzhang12345/os-homework